"Giuro solennemente

di non avere

buone intenzioni"

Felpato, Ramoso,

Codaliscia, Lunastorta

DOMANDE INGSOFT

Procedimento di compilazione ed esecuzione del codice all'interno del framework .NET tramite il CLR

Approfondimenti sui componenti .NET
http://www.deathlord.it/pro/sl/ricerca/tutorial.net/contents/2_Architettura/index.htm

Compilazione e esecuzione
Inizialmente il codice di alto livello viene precompilato da .NET nel linguaggio intermedio comune (l'IL) e salvato in memoria come file assembly.
All'interno del file precompilato sono presenti, oltre al codice IL, i metadati che lo descrivono (compresa la descrizione dei tipi di dato utilizzati al suo interno) ed eventuali risorse utilizzate (immagini, icone...).
Per fare eseguire il codice IL, questo deve essere compilato dal compilatore JIT (Just In Time) del CLR che, a differenza dei compilatori tradizionali, non compila l'intero codice ma converte delle parti dell'IL, al bisogno, durante l'esecuzione del programma stesso. Il codice compilato dal JIT viene salvato in memoria per essere reso disponibile ad altre future chiamate, evitando una molteplice compilazione della stessa porzione di codice.

CLR [FACOLTATIVO]
Il Common Language Runtime è l'implementazione proprietaria di Microsoft del Common Language Infrastructure; questo consente l'integrazione di applicazioni scritte in diversi linguaggi di alto livello, pre compilando le applicazioni in un linguaggio comune (l'Intermediate Language).
Poiché ogni linguaggio .NET deve aderire alle regole dettate dal framework (le Common Language Specification), la differenza nella rappresentazione dei tipi di dato (primitivi e non) tra i vari linguaggi viene risolta con la definizione di un sistema di tipi unificato e inter-linguaggio, il Common Type System.
Il CLR fornisce inoltre la gestione del ciclo di vita di tutti gli oggetti istanziati, del garbage collector (che non usa il Reference Counting, permettendo riferimenti circolari e velocizzando l'allocazione), di ereditarietà ed interfacce, del multithreading e della sicurezza.

Principio aperto/chiuso (Open/Closed Principle)

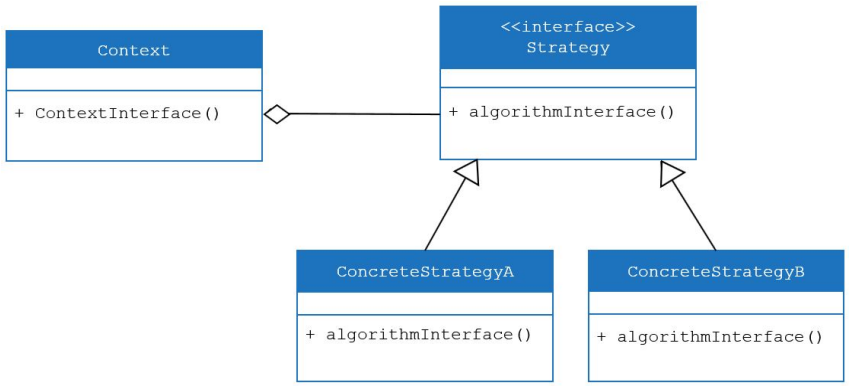
È un principio che permette di progettare delle entità fortemente riutilizzabili, pensandole *Open* alle estensioni (aggiunta di nuovi stati, comportamenti o proprietà) ma *Closed* alle modifiche.
In particolare l'estensione delle entità deve essere resa possibile senza modificare il codice già scritto, ciò viene reso possibile grazie a meccanismi di astrazione, quali interfacce o classi astratte. Poiché in questo principio un modulo viene definito Chiuso nel momento in cui viene reso utilizzabile da altri, la definizione dei suoi comportamenti deve essere ben definita, la sua interfaccia quindi non modificabile.
L'estensibilità del principio è quindi data dalla possibilità di aggiungere nuovi comportamenti nelle classi che implementano le interfacce così da rendere le modifiche trasparenti all'utilizzatore.

Pattern Strategy e Adapter con esempi

Pattern Strategy (behavioral - comportamentale)

Il pattern strategy viene utilizzato nei casi in cui viene definito un comportamento per il quale siano possibili diverse implementazioni e l'utilizzo di un'implementazione piuttosto che un'altra risulta indifferente all'utilizzatore.
Il pattern viene realizzato tramite l'utilizzo di una interfaccia, che rappresenterà il comportamento astratto, e dalle sue implementazioni, intercambiabili a runtime.

Esempio: Strategy -> Sorter; ConcreteA -> BubbleSort; ConcreteB-> Naive;
Context -> ListUtils

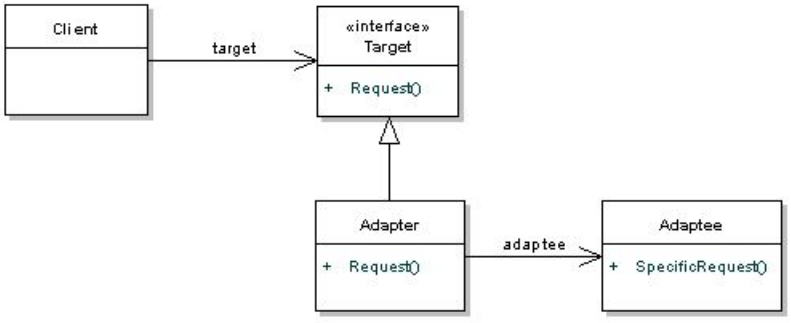


Pattern Adapter (structural - strutturale)

Il pattern adapter viene utilizzato per rendere compatibili due interfacce che di fatto non lo sarebbero, senza dover apportare modifiche al codice.
Il pattern viene realizzato tramite la creazione di una classe intermedia "Adapter" che si occupa di effettuare, per l'appunto, l'adattamento dalla classe nativa a quella desiderata.

- Target: Interfaccia desiderata dal Client
- Adapter: adatta l'interfaccia Adaptee a quella Target
- Adaptee: Interfaccia che deve essere adattata
- Client: Entità che utilizza solo interfacce di tipo Target

Esempio: Target -> Cane; Adapter -> CaneAdapter; Adaptee -> CaneRobot;
Request() -> Abbaia(); SpecificRequest() -> RiproduciSuonoAbbaio();
TargetImpl -> CaneVivente (Classe che implementa Target)



Capacità di sopravvivenza del sistema

La capacità di sopravvivenza del sistema è la proprietà di un sistema di poter continuare a fornire i suoi servizi anche mentre questo è sotto attacco o in stato di recupero da un danneggiamento interno.
Fasi di Analisi di Sopravvivenza:

1. **Capire il sistema:** analisi di obiettivi, requisiti e architettura del sistema
2. **Identificazione dei servizi critici**
3. **Simulare gli attacchi:** Identificare misuse case e possibili attacchi
4. **Analisi della sopravvivenza:** identificazione di componenti essenziali o a rischio e delle strategie di sopravvivenza

Strategie per garantire la capacità di sopravvivenza, per ognuna di esse è necessario implementare nel sistema dei componenti ad hoc:

- **Resistenza:** analisi su come evitare problemi interni al sistema (ricerca dei servizi critici) e come respingere attacchi
- **Identificazione:** capacità di riconoscimento di attacchi e fallimenti e valutazione del danno
- **Ripristino:** tolleranza del sistema durante gli attacchi e ripristino in seguito

Principali categorie di requisiti della sicurezza

Pari pari da pape

Tipi di requisiti:

- **Requisiti per l'integrità:** si deve garantire che le informazioni del prodotto software rimangano integre, cioè senza alterazioni di qualche natura, anche accidentale.
- **Requisiti per l'identificazione:** si ha la necessità di identificare l'utente per accedere a determinate parti del sistema (chi sei?). Esempi di identificazione possono essere un ID numerico, un username. La biometria è stata utilizzata come meccanismo di identificazione, anche se per la sua procedura di confronti, è più efficace come sistema di autenticazione.
- **Requisiti per l'autenticazione:** servono per autenticare l'utente dopo averlo identificato, cioè l'utente deve dimostrare chi dice di essere.
- **Requisiti di non ripudiabilità:** se un utente effettua un'operazione, l'utente non può negare di aver effettuato tale operazione. Il sistema di cifratura a chiave asimmetrica riesce a garantire (per ora) tale requisito, mediante l'utilizzo di calcoli computazionalmente difficili.
- **Requisiti di immunità:** si deve garantire una protezione contro le minacce esterne, per esempio virus, worm, etc.
- **Requisiti di autorizzazione:** servono per gestire i diritti dell'utente all'interno del prodotto software (cosa può fare?)
- **Requisiti di manutenzione:** servono per garantire che durante la manutenzione non vengano effettuate modifiche illecite che possano effettuare danni al sistema
- **Requisiti di scoperta delle intrusioni:** si deve cercare di scoprire se una minacce si è introdotta nel sistema. Per esempio si possono utilizzare dei log insieme a un sistema per analizzare tali log, come avviene nell'esempio Villaggio turistico mostrato a lezione.
- **Requisiti di protezione:** si deve garantire una certa sicurezza durante l'utilizzo del prodotto software.

Oltre a tali requisiti è consigliato seguire una serie di regole pratiche:

- **Validare tutti gli input:** la validazione consente di assicurare protezione da crash e exploit. Un pen tester entra in un bar: ordina 1 birra, ordina -1 birre, ordina 999999 birre, ordina NULL birre, ordina ; DROP tables -- birre.
- **Consapevolezza dell'ingegneria sociale**
- **Pianificare il deployment:** minimizzare le autorizzazioni a default, localizzare le configurazioni, fornire tool per le configurazioni, fornire strumenti per l'aggiornamento.
- **Pianificare il ripristino**
- **Bilanciare sicurezza e usabilità**
- **Compartimentare i beni:** i beni possono essere divisi, per esempio utilizzando l'architettura a layer o attraverso l'architettura Client-Server, con i vari privilegi e difetti.

Single Responsibility Principle

Il single responsibility principle si basa sulle premesse che:

- ogni classe debba implementare una singola funzionalità del programma, completamente, correttamente e sia l'unica a farlo
- non debba esistere più di un motivo per effettuare delle modifiche sulla classe

L'applicazione del principio consente di sviluppare classi indipendenti tra loro, semplici e riutilizzabili; in caso, invece, una classe abbia più responsabilità queste diventano accoppiate e modifiche apportate ad una di esse richiedono modifiche anche sull'altra, portando allo sviluppo di oggetti fragili.

Esempio:
La classe StampanteDaUfficio ha due responsabilità indipendenti tra loro
(stampa di un file e scansione), la classe risulta essere poco portabile in contesti nei quali sia necessaria una semplice stampante o uno scanner.
Per questo, applicando il SRP, la classe viene separata in:

Eventuali modifiche apportate su una delle due classi non avrà ripercussioni sull'altra.

Vantaggi e svantaggi del modello Copy-Modify-Merge

Nella gestione del versionamento il modello Copy-Modify-Merge propone una soluzione nella quale non sono presenti meccanismi di blocco delle risorse; per effettuare una modifica su un elemento se ne crea una copia, si lavora sulla copia e al momento del check-in si uniscono le modifiche effettuate sulla copia (operazione di *merge*) al documento originale.
Il merge delle versioni può generare, o meno, conflitti; questi sono presenti nel caso sia rilevante l'ordine con il quale vengono apportate le modifiche.
Vantaggi:

- Non è necessario conoscere gli utenti ma solo le modifiche che questi hanno apportato
- Ogni utente ha una sua copia proprietaria sulla quale effettuare le modifiche, è quindi possibile lavorare parallelamente
- Effettuare il merge tra due versioni nelle quali sono state modificate righe di codice diverse non produce conflitti

Svantaggi:

- È possibile effettuare modifiche solo su una copia dell'ultima versione disponibile, in caso contrario non sarà possibile apportare con successo nuove modifiche
- Non è possibile rilevare a priori dei conflitti semantici, il merging potrebbe quindi andare a buon fine ma la nuova versione potrebbe non essere funzionante
- In caso vengano modificate parallelamente le stesse righe di codice, il conflitto deve essere risolto manualmente, in quanto il VCS non è in grado di decidere la modifica adatta

Differenza tra tipi valore e tipi riferimento in .NET

Le variabili di tipo riferimento archiviano i riferimenti ai relativi dati (oggetti), mentre le variabili dei tipi di valore contengono direttamente i dati.
I tipi valore rappresentano generalmente i tipi primitivi (ma anche ***struct*** ed ***enum***) e non possono assumere valori **null**; i tipi riferimento, invece, rappresentano tutti quei dati espressi tramite la dichiarazione di classi per descrivere oggetti complessi e possono assumere valore **null**.
I tipi valore sono allocati sullo **stack**, mentre i tipi riferimento sono allocati nella memoria **managed heap**. Questi sono rappresentati da un **puntatore** (allocato nello stack) che fa riferimento ad una **locazione di memoria** dove è contenuto il valore vero e proprio. Mentre i tipi valore sono sequenze di byte, i tipi riferimento sono indirizzi di memoria.
Due variabili di tipo riferimento possono fare riferimento allo stesso oggetto. Invece, ogni variabile di tipo valore ha una propria copia dei dati e le operazioni su una variabile non influiscono su un'altra (tranne nel caso di variabili parametri "in", "out" e "ref").
Ogni tipo valore può essere automaticamente convertito, mediante up cast, in tipo riferimento (**boxing**):

```
double d = 123.4;
object o = d;
```

Il processo inverso del boxing e l'**unboxing** e permette, tramite down cast, di convertire un tipo riferimento in tipo valore:

```
double k = (double) o;
```

Principi per l'architettura dei package

I **principi di architettura dei package** sono:

- **REP (Reuse/Release Equivalency Principle):** un componente riutilizzabile (classe o cluster di classi) non può essere riutilizzato a meno che non venga gestito da un sistema di rilascio. Un cliente dovrebbe riutilizzare un componente soltanto se l'autore promette di mantenere traccia del numero di versione e mantenere vecchie versioni (per qualche tempo). La granularità del riuso è la stessa del rilascio
- **CCP (Common Closure Principle):** classi che cambiano insieme, devono stare nello stesso package. Si vuole minimizzare il numero dei package che vengono cambiati in un determinato ciclo di rilascio del prodotto.
- **CRP (Common Reuse Principle):** classi che tendono ad essere riutilizzate insieme vanno nello stesso package. Una dipendenza dal package implica una dipendenza da tutto ciò che si trova all'interno del package.

Queste tre architetture non possono essere soddisfatte contemporaneamente. Le architetture REP e CRP rendono facile la riusabilità, mentre il CCP il mantenimento. Inoltre, il CCP mira a rendere i package il più grande possibile, mentre il CRP il più piccolo possibile. L'architettura CCP è consigliata per le prime fasi di un progetto, mentre REP e CRP sono consigliati per un'architettura consolidata.

Principi per le relazioni tra i package

I **principi per le relazioni tra i package** sono i seguenti:

- **ADP (Acyclic Dependencies Principle):** le dipendenze tra i package **non devono essere cicliche**. Infatti, anche una sola dipendenza ciclica potrebbe rendere la lista delle dipendenze molto lunga. Per questo, una volta modificato un package, gli sviluppatori possono integrarlo nel resto del progetto, ma solo dopo averlo testato; per testarlo bisogna compilarlo insieme a tutti i package da cui dipende
- **SDP (Stable Dependencies Principle):** un package dovrebbe dipendere soltanto da package più stabili di lui. Si può ottenere questo risultato conformandosi al CCP. Alcuni package sono volutamente volatili, perché ci si aspetta che cambino. Un package da cui molti dipendono è fortemente stabile, in quanto richiede molto lavoro per riconciliare qualsiasi cambiamento con tutti i package dipendenti
- **SAP (Stable Abstractions Principle):** package stabili **devono** essere **package astratti**. La stabilità è correlata alla quantità di lavoro richiesto per effettuare una modifica

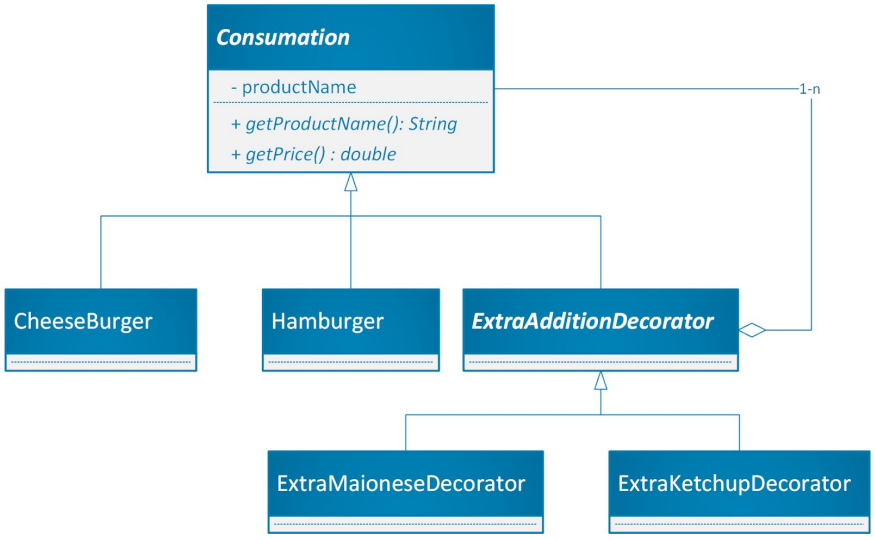
In generale, considerando una gerarchia a livello dei package, quelli che si trovano in cima sono instabili e flessibili, mentre quelli che si trovano alla base sono stabili, perché difficili da modificare. Tuttavia, seguendo l'open/close principle, i package della base devono essere facilmente estendibili, quindi è richiesto un alto livello di astrazione.
Pertanto, è possibile creare un'applicazione composta da package instabili, ma facilmente modificabili, e package stabili, facilmente estendibili.

Pattern Decorator con esempi (structural - strutturale)

Il pattern Decorator permette di aggiungere dinamicamente responsabilità ad un oggetto. Fornisce un'alternativa flessibile alla specializzazione, da preferire quando le possibili estensioni sono talmente numerose da rendere impossibile supportare ogni combinazione (bisognerebbe definire un elevato numero di sottoclassi).
Struttura di un decorator:

- **Component** (interfaccia o classe astratta): dichiara un'interfaccia comune a tutti gli oggetti a cui verranno aggiunte responsabilità
- **ConcreteComponent:** definisce un tipo di oggetto al quale è possibile aggiungere dinamicamente responsabilità
- **Decorator** (classe astratta): mantiene un riferimento ad un oggetto di tipo Component e definisce un'interfaccia conforme all'interfaccia Component
- **ConcreteDecorator:** aggiunge funzionalità al Component

Esempio:



Una paninoteca vende panini identificati da nome e prezzo. Nell'esempio ci sono due tipi di panino: Hamburger e CheeseBurger; essi non hanno al loro interno maionese o ketchup. Se il cliente vuole aggiungere uno o più ingredienti extra deve pagare una somma aggiuntiva.

- **Component:** classe "Consumation", che modella la gerarchia di consumazione del cliente
- **Decorator:** classe "ExtraAdditionDecorator", che modello ogni possibile aggiunta ad un prodotto
- **ConcreteComponent:** entità del dominio, ovvero i prodotti venduti dalla paninoteca: "Hamburger" e "CheeseBurger"
- **ConcreteDecorator:** decorator concreti, nell'esempio due tipi di ingredienti "ExtraKetchupDecorator" e "ExtraMaioneseDecorator"

Spiegare il modello a cascata e il modello iterativo

Modello a cascata:
Secondo questo modello, poiché introdurre nel software sostanziali cambiamenti in fasi avanzate dello sviluppo ha costi elevati, per non generare retroazioni bisogna svolgere esaustivamente ogni fase prima di passare alla successiva.
Il prodotto di ogni fase (ovvero l'ingresso della fase successiva) prende il nome di **semilavorato**.
Questo modello è limitato dalla sua rigidità, in quanto devono sussistere:

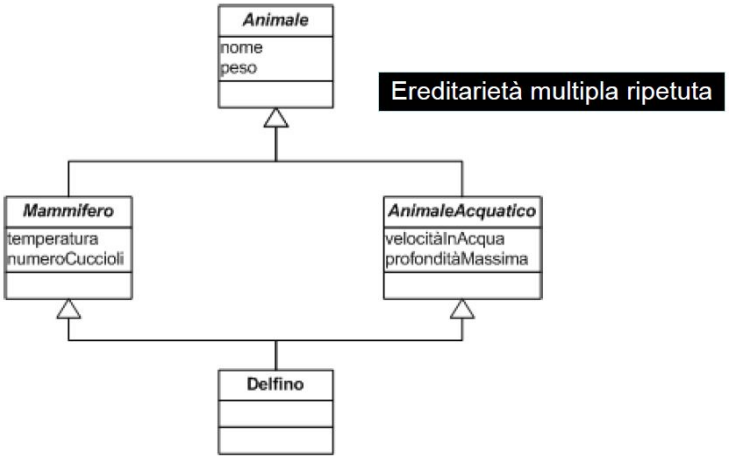
- **Immutabilità dell'analisi:** in fase di analisi deve essere possibile definire tutte le funzionalità che il software andrà a realizzare
- **Immutabilità del progetto:** è possibile progettare l'intero sistema prima ancora di iniziare l'implementazione

In un modello reale questi requisiti non possono essere soddisfatti, in quanto le specifiche cambiano in continuazione. Per questo si tende a realizzare prima un prototipo e poi il sistema reale secondo il modello a cascata.

Modello iterativo:
L'idea alla base di questo metodo è quello di sviluppare un sistema attraverso cicli ripetuti, andando ad aumentare, ad ogni iterazione, il livello di dettaglio del sistema stesso.
Di difficile implementazione per progetti significativi.

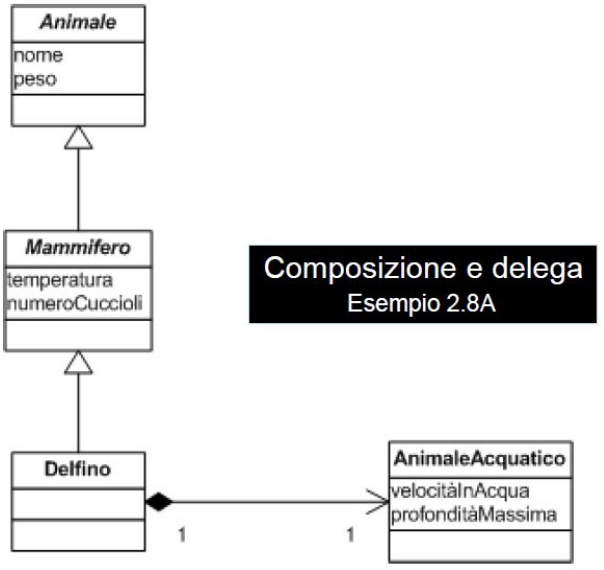
Come si implementa l'ereditarietà multipla

Se abbiamo strutture con ereditarietà multipla e il linguaggio di programmazione che usiamo non la supporta, è necessario convertire le strutture con ereditarietà multipla in strutture con solo ereditarietà semplice.

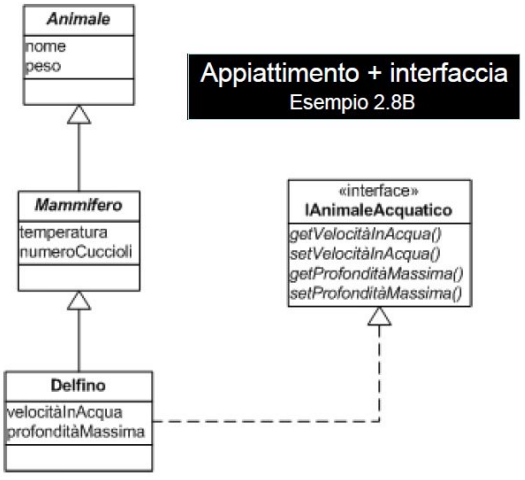


Per fare questo abbiamo più di una possibilità:

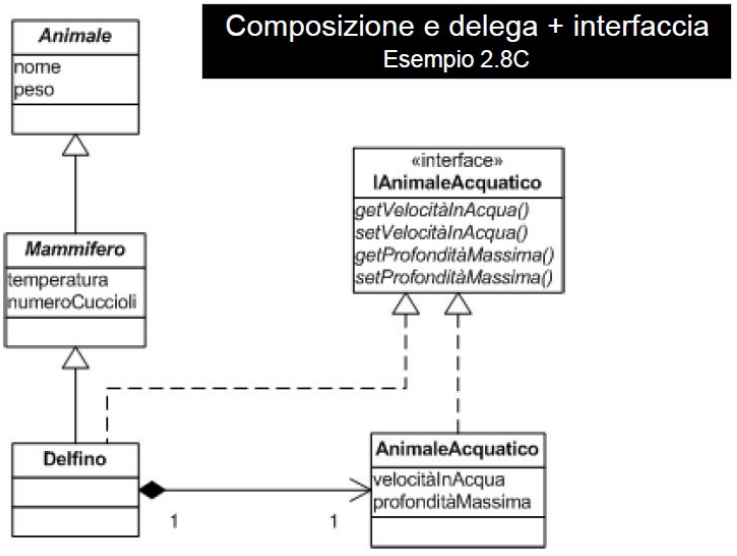
1. **Composizione e delega:**
   - Si sceglie la più significativa fra le superclassi e si fa ereditare solo da quella
   - Le rimanenti superclassi diventano possibili ruoli e vengono connesse mediante composizione
   - In questo modo le caratteristiche delle superclassi escluse vengono incorporate nella classe specializzata tramite composizione e delega e non tramite ereditarietà.



2. **Interfacce:**
   Viene appiattito il tutto in una gerarchia semplice e implementate le caratteristiche restanti come interfaccia. In questo modo, una o più relazioni di ereditarietà si perdono e gli attributi e le operazioni corrispondenti devono essere ripetute nelle classi specializzate



3. **Una combinazione delle precedenti:**
   Composizione e delega e l'uso di interfacce possono essere combinati per risolvere l'ereditarietà multipla



Tipi di dati in .NET

Intro

Common Type System definisce le modalità di dichiarazione, utilizzo e gestione dei tipi in Common Language Runtime e rappresenta una parte importante del supporto runtime per l'integrazione di più linguaggi. Le funzioni assolte dal sistema di tipi comuni sono le seguenti:

- Definire un framework che consenta l'integrazione di più linguaggi, l'indipendenza dai tipi e l'esecuzione di codice con prestazioni elevate.
- Fornire un modello orientato a oggetti che supporta l'implementazione completa di molti linguaggi di programmazione.
- Definire le regole che i linguaggi devono seguire, garantendo l'interazione tra oggetti scritti in linguaggi diversi.
- Fornire una libreria che contiene i tipi di dati primitivi, ad esempio Boolean, Byte, Char, Int32 e UInt64, utilizzati nello sviluppo delle applicazioni.

Tipi in .NET

Tutti i tipi in .NET sono tipi valore o tipi riferimento.

- **Tipi valore:** sono tipi di dati i cui oggetti sono rappresentati dal valore effettivo dell'oggetto. Se un'istanza di un tipo di valore viene assegnata a una variabile, a tale variabile viene fornita una copia aggiornata del valore. Sono allocati sullo **stack** e sono una sequenza di byte. Non possono avere valore null.
- **Tipi riferimento:** sono tipi di dati i cui oggetti sono rappresentati da un riferimento (simile a un puntatore) al valore effettivo dell'oggetto. Se un tipo di riferimento viene assegnato a una variabile, tale variabile fa riferimento (punta) al valore originale. Non viene effettuata alcuna copia. Sono riferimenti a oggetti allocati sull'**heap**. Possono avere valore null.

Common Type System in .NET supporta cinque categorie di tipi (Classi, Strutture, Enumerazioni, Interfacce, Delegati).

[LUCO]In generale, un tipo può contenere la definizione di 0+...
[LUCO]Modificatori di visibilità

Spiegare i quattro bad design (fragilità, immobilità, rigidità, viscosità)

1. La **rigidità** è la tendenza del software a essere difficile da cambiare. Ogni modifica causa una serie di modifiche a cascata nei moduli dipendenti e quindi per una piccola modifica è necessario modificare gran parte del programma. L'effetto di questo problema è che si tende a posporre la risoluzione di problemi non critici per paura di perdere troppo tempo ad apportare le modifiche necessarie al resto del codice.
2. La **fragilità** è la tendenza del software a rompersi in molti punti ogni volta che viene cambiato; i cambiamenti tendono a causare comportamenti imprevisti in altre parti del sistema (spesso in aree che non hanno relazioni concettuali con l'area che è stata modificata). Ogni correzione peggiora le cose, introducendo più problemi di quanti ne vengano risolti, quindi tale software è impossibile da mantenere; come effetto si ha che ogni volta che si fa una correzione, si teme che il software si rompa in modo imprevisto.
3. L'**immobilità** è l'incapacità di riutilizzare il software da parte di altri progetti o da altre parti dello stesso progetto. Ad esempio, uno sviluppatore scopre di aver bisogno di un modulo simile a quello scritto da un altro sviluppatore, ma il modulo in questione si collega troppo ad altre parti del software. Dopo molto lavoro, lo sviluppatore scopre che il lavoro e il rischio necessari per separare le parti desiderabili del software da quelle non desiderate sono troppo grandi da tollerare, quindi non ne vale la pena. Questo si tramuta nel riscrivere da capo il software al posto di riutilizzarlo.
4. Per quanto riguarda la **viscosità**, gli sviluppatori di solito trovano più di un modo per apportare una modifica: alcuni preservano il design, altri no (ovvero sono hack). Si ha quindi la tendenza a incoraggiare i cambiamenti del software che sono hack piuttosto che cambiamenti del software che preservano l'intento di progettazione originale.
   Esistono due tipi di viscosità:

   - **viscosità del design**: si verifica quando implementare entità rispettando il design è più difficile che farlo usando dei workaround
   - **viscosità dell'ambiente**: l'ambiente di sviluppo è lento e inefficiente (i tempi di compilazione sono molto lunghi, il sistema di test è lento, il version control complesso)

Si ha quindi che è facile fare la cosa sbagliata ed è difficile fare la cosa giusta: questo comporta una degenerazione della manutenibilità del software a causa di hack, scorciatoie, fix temporanei, etc.

Principio di sostituibilità di Liskov

Il principio di sostituibilità di Liskov afferma che:
“B è una sotto-classe di A se e solo se ogni programma che utilizzi oggetti di classe A può utilizzare oggetti di classe B senza che il comportamento logico del programma cambi”
Il principio di sostituibilità di Liskov viene utilizzato come guida per creare mediante ereditarietà le classi concrete descritte nell’Open-Close Principle.
Per evitare violazioni al principio di Liskov si ricorre al Design by Contract, secondo il quale ogni metodo ridefinito in una sottoclasse deve avere:

- pre-condizioni identiche o meno stringenti rispetto alla superclasse
- post-condizioni identiche o più stringenti rispetto alla superclasse

Inoltre nella sottoclasse:

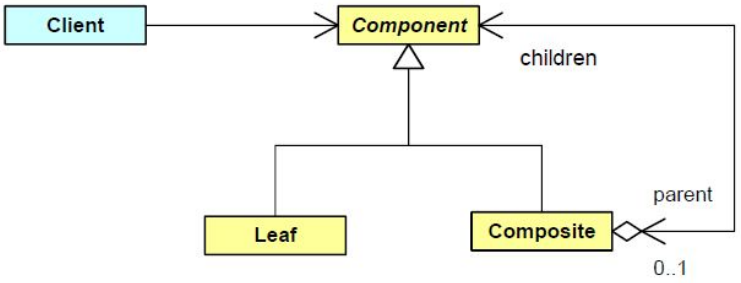
- la semantica della classe base deve essere mantenuta
- non è possibile aggiungere vincoli comportamentali rispetto alla classe base

Per fare un esempio in cui il principio di sostituibilità di Liskov non viene rispettato, si pensi a una classe Quadrato e una classe Rettangolo. Il Quadrato è un Rettangolo ma aggiunge un vincolo: quando viene modificata l’altezza viene modificata anche la larghezza. Ciò significa che una modifica a una dimensione comprende un cambiamento nello stato dell’oggetto che non avverrebbe invece utilizzando un’istanza della superclasse. Dunque un Quadrato non può essere sostituibile secondo Liskov a un Rettangolo.

Pattern Composite con esempi **(structural - strutturale)**

Pattern Composite

Permette di comporre oggetti in una struttura ad albero, al fine di rappresentare una gerarchia di oggetti contenitori-oggetti contenuti, permettendo ai clienti di trattare in modo uniforme oggetti singoli e oggetti composti.



È composto da:

- **Component** (classe astratta) dichiara l'interfaccia e realizza il comportamento di default
- **Client** accede e manipola gli oggetti della composizione attraverso l'interfaccia di Component
- **Leaf** descrive oggetti che non possono avere figli-foglie e definisce il comportamento di tali oggetti
- **Composite** descrive oggetti che possono avere figli-contenitori e definisce il comportamento di tali oggetti.

Il contenitore dei figli deve essere un attributo di Composite e può essere di qualsiasi tipo (array, lista, albero, tabella hash, etc.). È presente un riferimento esplicito al genitore (parent): questo semplifica l'attraversamento e la gestione della struttura, ma l'attributo che contiene il riferimento al genitore e la relativa gestione devono essere posti nella classe Component. Tutti gli elementi che hanno come genitore lo stesso componente devono essere gli unici figli di quel componente.

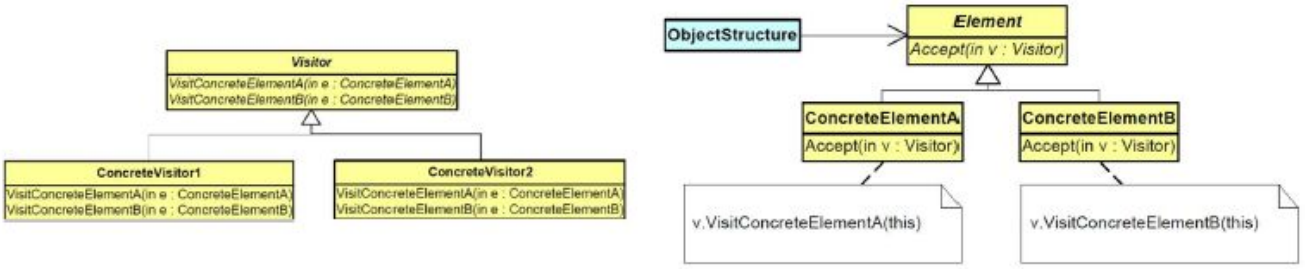
Gli obiettivi del pattern composite sono:

- **massimizzazione dell'interfaccia Component**: si fa in modo che il cliente veda solo l'interfaccia di Component.
- **trasparenza**: dichiarando tutto al livello più alto, il cliente può trattare gli oggetti in modo uniforme.
- **sicurezza**: tutte le operazioni sui figli vengono messe in Composite. A questo punto, qualsiasi invocazione sulle foglie genera un errore in fase di compilazione ma il cliente deve conoscere e gestire due interfacce differenti. Nel caso si scelga come obiettivo principale la sicurezza, occorre disporre un modo per verificare se l'oggetto su cui si vuole agire è un Composite.

Esempio: Eserciti. Molti eserciti sono composti da divisioni, a loro volta composte da platoni e squadre, fino ai singoli soldati. Gli ordini vengono dati dalla cima della gerarchia a scendere finchè ogni soldato sa ciò che deve fare.

Pattern Visitor con esempi **(behavioral - comportamentale)**

Permette di definire una nuova operazione da effettuare sugli elementi di una struttura senza dover modificare le classi degli elementi coinvolti. Tutto il codice relativo ad un singolo tipo di operazione viene raccolto in una singola classe e per aggiungere un nuovo tipo di operazione è sufficiente progettare una nuova classe. Il visitor deve dichiarare un'operazione per ogni tipo di nodo concreto e ogni nodo deve dichiarare un'operazione per accettare un generico visitor.



- **Visitor**, una classe astratta o un'interfaccia, dichiara un metodo Visit per ogni classe di elementi concreti mentre ConcreteVisitor definisce tutti i metodi Visit e globalmente definisce l'operazione da effettuare sulla struttura e, se necessario, ha un proprio stato.
- **Element**, una classe astratta o un'interfaccia, dichiara un metodo Accept che accetta un Visitor come argomento e ConcreteElement definisce il metodo Accept. ObjectStructure può essere realizzata come Composite o come una normale collezione (array, lista, etc.); deve poter enumerare i suoi elementi e deve dichiarare un'interfaccia che permetta a un cliente di far visitare la struttura a un Visitor.

Nel complesso, **il pattern visitor facilita l'aggiunta di nuove operazioni**. È possibile aggiungere nuove operazioni su una struttura esistente semplicemente aggiungendo un nuovo visitor concreto; senza il pattern visitor sarebbe necessario aggiungere un metodo ad ogni classe degli elementi della struttura. Ogni Visitor concreto raggruppa i metodi necessari per eseguire una data operazione e nasconde i dettagli di come tale operazione debba essere eseguita.
È difficile aggiungere una nuova classe ConcreteElement in quanto per ogni nuova classe ConcreteElement è necessario inserire un nuovo metodo Visit in tutti i Visitor esistenti: la **gerarchia Element** deve essere poco o per nulla modificabile, ovvero **deve essere stabile**.

Polimorfismo secondo Cardelli-Wegner, Polimorfismo per inclusione

Il polimorfismo è la capacità di uno stesso elemento di assumere forme diverse in contesti diversi, o di elementi diversi di assumere la stessa forma in un determinato contesto. La classificazione Cardelli-Wegner dei polimorfismi li divide in due macro-categorie: *universale* e *ad-hoc*.

- ***Universale*** - Può assumere un numero indefinito di forme
    - *Per inclusione*: nella programmazione orientata agli oggetti è rappresentato dall'overriding dei metodi e dal conseguente meccanismo di late binding tra il metodo e la sua implementazione.
    - *Parametrico*: programmazione generica rispetto ai tipi, come le classi generiche dove il tipo di una o più variabili è un parametro della classe stessa. Verranno generate classi indipendenti senza alcun rapporto di ereditarietà (`List<Element>, Comparable<Studente>`)
- ***Ad-hoc*** - Può assumere un numero definito di forme
    - *Overloading*: ridefinizione di metodi/operatori per ogni insieme di argomenti accettati → definizione in fase di programmazione
    - *Coercion*: una variabile di un certo tipo viene convertita (implicit. o esplicit.) a un tipo diverso → le varie conversioni possibili devono essere definite in fase di programmazione

Passaggio dei parametri in C#

Il passaggio degli argomenti in C# può avvenire in 3 diverse forme:

1. **in** (default): l'argomento deve essere inizializzato, viene passato per valore (copiato) ed eventuali modifiche al valore dell'argomento non hanno effetto sul chiamante;
2. **in/out** (`ref`): l'argomento deve essere inizializzato, viene passato per riferimento ed eventuali modifiche del valore dell'argomento hanno effetto sul chiamante;
3. **out** (`out`): l'argomento può non essere inizializzato, viene passato per riferimento e le modifiche del valore dell'argomento (l'inizializzazione è obbligatoria) hanno effetto sul chiamante.

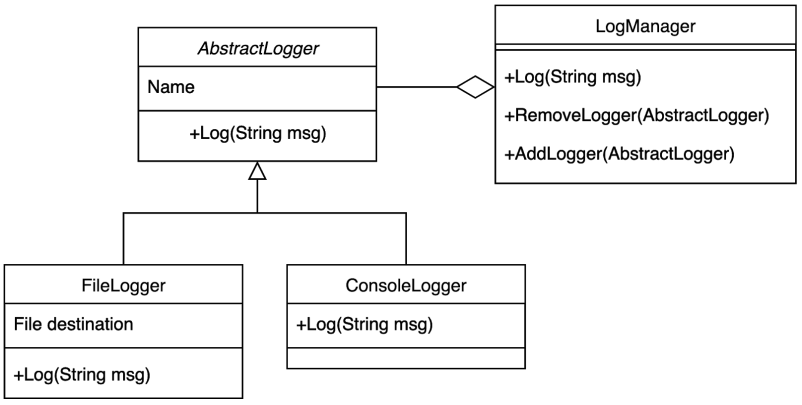
```
public static void Main()
{
    string a = "Stringa iniziale";
    caso1(a);
    Console.WriteLine(a);

    string b = "Stringa iniziale";
    caso2(ref b);
    Console.WriteLine(b);

    string c;
    caso3(out c);
    Console.WriteLine(c);
}

public static void caso1(string x)
{
    x = "Ciaooo";
}

public static void caso2(ref string x)
{
    x = "Stringa modificata";
}

public static void caso3(out string x)
{
    x = "Stringa inizializzata";
}
```

Le regole da rispettare sono: utilizzare prevalentemente il passaggio standard per valore e utilizzare il passaggio per riferimento (`ref` o `out`) solo se strettamente necessario, ad esempio quando è necessario restituire al chiamante più di un valore. In questo caso occorre scegliere `ref` se l'oggetto passato come argomento deve essere già stato inizializzato prima della chiamata e `out` se è responsabilità del metodo inizializzare completamente l'oggetto passato come argomento.

Pattern Observer con esempi **(behavioral - comportamentale)**

Il pattern **observer** risolve il problema dell'accoppiamento forte tra elementi dipendenti che devono essere notificati circa una modifica avvenuta su uno dei due. In particolare un elemento, il **subject,** deve notificare tutti gli **observer** che si sono registrati presso di lui, ogni qualvolta una determinata azione ne cambia lo stato. Scrivere la logica di aggiornamento nel **subject** porterebbe ad un accoppiamento forte tra le parti, e sarebbe quindi una soluzione poco manutenibile.

Una possibile soluzione è quindi quella di impostare una relazione uno a molti tra soggetto e osservatori. Quindi ogni oggetto **subject** avrà al suo interno una collezione di generici oggetti **observer** sui quali invocherà un metodo prestabilito ogni qualvolta avviene una modifica saliente. La classe **subject** dovrà inoltre esporre i metodi di *registrazione* e *deregistrazione* per consentire agli oggetti **observer** di aggiungersi/rimuoversi dalla collezione del subject.



Il pattern observer è una delle 5 diverse soluzioni per il problema dell'aggiornamento di elementi che "osservano" lo stato di un soggetto. Gli altri 4 sono:

- Class-based - Ogni subject ha un riferimento diretto all'observer da notificare
- Interface-based - Ogni subject ha un riferimento all'interfaccia che l'observer implementa, in modo da disaccoppiare subject/observer
- Delegate-based - Invece che un riferimento ad un'interfaccia, il subject contiene un **delegato pubblico** in modo che gli observer si possano registrare a tale delegato e venire notificati indirettamente dal subject
- Event-based - Come il delegate-based ma non espone pubblicamente il delegato, bensì un **event** che si occupa di (de)registrare i vari observers

Nell'esempio a lato, l'oggetto LogManager è il **subject** che, una volta invocato il metodo *Log*, notificherà tutti i suoi **observers** di tipo *AbstractLogger,* ognuno con una destinazione diversa per il messaggio di log.

Modello Lock-Modify-Unlock nei VCS con vantaggi e svantaggi

Il modello Lock-Modify-Unlock (LMU) è un modello utilizzato dai VCS centralizzati per minimizzare i rischi di sovrascrittura del lavoro altrui sugli stessi file. In particolare il sistema tiene traccia di tutti i file che sono correntemente in fase di modifica da un utente e ne blocca ogni altra possibilità di modifica (mettendo quindi un *lock* a tali file); solo quando l'utente iniziale termina le modifiche, il sistema rilascerà il lock, permettendo ad altri utenti di acquisirlo. Questa strategia, essendo particolarmente semplice, presenta svariati problemi:

- Possibilità di rallentare tutto il team di sviluppo in caso di mancato rilascio del lock da parte di un utente (magari per dimenticanza);
- Serializzazione del lavoro superflua: se due utenti vogliono modificare parti diverse di un file, nonostante queste modifiche non interferiscano in alcun modo l'una con l'altra, il sistema lo impedisce;
- Falso sentimento di sicurezza: il fatto che due utenti modifichino file diversi non significa che le loro modifiche siano necessariamente indipendenti;
- Richiede agli sviluppatori di essere costantemente online.

Regole di sicurezza nella progettazione (linee guida)

1. *Basare le decisioni della sicurezza su una politica esplicita*
   Le politiche di sicurezza sono documenti di alto livello che definiscono cosa sia la sicurezza ma non come ottenerla. ***[EXTRA]***Le politiche di sicurezza devono essere incorporate nella progettazione al fine di:
   - specificare in che modo le informazioni possono essere accedute e chi può accedervi;
   - quali precondizioni devono essere validate per l'accesso;
2. *Evitare un singolo punto si fallimento*
   È necessario evitare che il fallimento di una parte del sistema si trasformi nel fallimento di tutto il sistema.
3. *Fallire in modo certo*
   I sistemi critici per la sicurezza dovrebbero sempre fallire in modo controllato.
4. *Bilanciare sicurezza e usabilità*
   L'aggiunta di controlli per la sicurezza ricade inevitabilmente sull'utente che ha bisogno di più tempo per imparare ad utilizzare il sistema
5. *Essere consapevoli dell'esistenza dell'ingegneria sociale*
   L'ingegneria sociale si occupa di trovare modi per convincere con l'inganno utenti accreditati al sistema a rivelare informazioni riservate.
6. *Usare ridondanza e diversità riduce i rischi*
   Ridondanza = mantenere più di una versione del software e dei dati nel sistema.
   Diversità = le diverse versioni del sistema non dovrebbero essere basati sulle stesse tecnologie così che una vulnerabilità della tecnologia non conduca a un comune punto di fallimento.
7. *Validare tutti gli input*
   È necessario definire nei requisiti tutti i controlli da applicare per impedire l'ingresso all'interno del sistema di input malevoli (*buffer overflow*, *SQL injections*)
8. *Dividere in compartimenti i beni*
   Dividere in compartimenti significa organizzare le informazioni nel sistema in modo che gli utenti abbiano accesso solo alle informazioni che competono loro.
9. *Progettare per il deployment*
   Progettare il sistema in modo che siano inclusi programmi di utilità per semplificare il deployment. Bisogna sempre progettare il sistema con l'assunzione che gli errori di sicurezza possano accadere.***[EXTRA]*** Alcune linee guida:
   - includere supporto per visionare ed analizzare le configurazioni;
   - minimizzare i privilegi di default;
   - localizzare le impostazioni di configurazione;
   - fornire modi per rimediare a vulnerabilità di sicurezza.
10. *Progettare per il ripristino*
    È necessario includere meccanismi diretti (eventualmente automatici) per aggiornare il sistema e riparare le vulnerabilità di sicurezza che vengono scoperte

RUP – Rational Unified Process

Rational Unified Process: modello di sviluppo ibrido iterativo. Definisce uno schema generale adattabile a più situazioni anziché definire uno schema preciso. Individua più prospettive:

Prospettiva dinamica: mostra le fasi nel tempo

- **Avvio**: analisi di fattibilità calata nel contesto di un preciso mercato e di uno specifico caso d'uso (chi interagirà col sistema). Permette di ottenere una prima valutazione dei rischi e dei requisiti.
- **Elaborazione**: definisce la struttura complessiva del sistema, completa di analisi del dominio e di una prima fase di progettazione dell'architettura. Terminata questa fase sarà poi più difficile e costoso fare modifiche.
- **Costruzione**: progettazione e programmazione in parallelo delle varie parti del sistema. Questa fase comprende anche la verifica delle singole parti isolate e integrate.
- **Transizione**: fase in cui il sistema passa dall'ambiente di sviluppo a quello del cliente. Bisogna controllare che il sistema sia conforme alle specifiche del cliente e istruire gli utenti finali all'uso del prodotto.

Prospettiva statica: si concentra sui workflow (attività di produzione). All'interno di una fase dinamica possono essere attivi anche tutti i workflow. RUP definisce 9 workflow (6 principali e 3 di supporto):

- Modellazione delle attività aziendali (P)
- Requisiti (P)
- Analisi e progetto (P)
- Implementazione (P)
- Test (P)
- Rilascio (P)
- Gestione della configurazione e delle modifiche (S)
- Gestione del progetto (S)
- Ambiente (S)

Prospettiva pratica: descrive delle buone prassi da usare nell'ingegneria del software:

- Sviluppare software ciclicamente: pianificare gli incrementi e sviluppare le funzioni secondo la loro priorità
- Gestire i requisiti: documentare esplicitamente i requisiti ed i cambiamenti effettuati analizzando l'impatto che i cambiamenti potrebbero avere
- Usare architetture basate sui componenti
- Creare modelli visivi del software
- Verificare la qualità del prodotto
- Controllare le modifiche del software con strumenti adatti

Tipologie di analisi dei requisiti (requisiti della sicurezza)

- **Analisi del rischio:** valutare le possibili perdite che un attacco può causare ai beni di un sistema e di bilanciare queste perdite con i costi richiesti per la protezione dei beni stessi. Le politiche di sicurezza propongono le condizioni che dovrebbero sempre essere mantenute dal sistema di sicurezza, quindi aiutano a identificare le minacce che potrebbero sorgere.
- **Analisi dei beni:**
    - **analisi risorse fisiche**: collocazione, quantità, collegamenti fisici
    - **analisi risorse logiche**: classificare le informazioni in base al valore che hanno per l'organizzazione, il grado di riservatezza e il contesto di afferenza
    - **analisi delle dipendenze tra risorse**: per ciascuna risorsa (fisica o logica) occorre individuare di quali altre risorse essa ha bisogno per funzionare correttamente.
- **Identificazione delle minacce:** definire quello che non deve poter accadere nel sistema, come attacchi intenzionali o eventi accidentali. In queste categorie ricadono ad esempio furti, danneggiamenti, tentativi di infiltrazione, guasti e perdite di credenziali suddivisi per intenzionalità e tipo di risorsa attaccata.
- **Valutazione dell'esposizione:** a ogni minaccia occorre associare un rischio così da indirizzare l'attività di individuazione delle contromisure verso le aree più critiche
- **Identificazione del controllo**: scegliere il controllo da adottare per neutralizzare gli attacchi individuati, in relazione al rapporto costo/efficacia

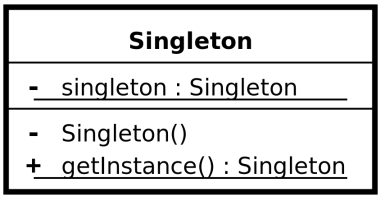
White box e black box testing

Il collaudo delle funzionalità di un sistema, così come quello della sua sicurezza, può avvenire seguendo un approccio a black box e/o a white box.

- **Black box** è mirata a collaudare le proprietà del sistema che sono visibili dall'esterno e che possono essere apprezzare senza conoscere il funzionamento del software: affidabilità, semplicità d'uso, velocità ecc. Nel caso dei test di sicurezza, il collaudatore si mette nei panni dell'attaccante e cerca di penetrare nel sistema sfruttando le conoscenze che si possono avere semplicemente guardando il sistema.
- **White box** invece è mirata ad analizzare la struttura interna del software e valutare caratteristiche come modularità e leggibilità del codice. Queste caratteristiche sono un modo per realizzare le proprietà esterne e ne influenzano la qualità. Nel caso dei test di sicurezza, il test white box consiste nell'analisi del codice e nella scrittura di test molto specifici che sfruttano le caratteristiche del codice per violare il sistema.

Pattern Singleton con esempi **(creational - comportamentale)**

Problema: Il pattern Singleton viene utilizzato quando è necessario garantire l'esistenza di al più una istanza di una classe, alla quale deve essere possibile accedere. Se esiste già una istanza di quella classe, allora è necessario intercettare tutte le richieste di creazione di una nuova istanza.
Soluzione: Si rende privato o protected il costruttore della classe. All'interno della classe si inserisce un membro statico con lo stesso tipo della classe. Si definisce un metodo statico `GetInstance()` che crea una istanza se non è già stata creata e la restituisce.
Conseguenze: non si può chiamare `new Singleton()` e si accede all'istanza solo tramite `GetInstance()`.
Una classe statica con solo membri statici non è una buona alternativa al pattern singleton, perché quest'ultimo permette di specializzare il metodo `GetInstance()` e restituire istanze specializzate, a seconda del contesto. Inoltre, una classe singleton non statica può implementare un numero arbitrario di interfacce.



Esempio di codice C#:

```
public class Singleton {
      … attributi membro di istanza …
      private static Singleton _instance = null;
      protected Singleton()
      { inizializzazione istanza }
      public static Singleton GetInstance()
      {
            if(_instance == null)
                  _instance = new Singleton();
            return _instance;
      }
       … metodi pubblici, protetti e privati …
}
```

Spiegare il modello a cascata e le sue criticità

Il modello a cascata si fonda sul presupposto che introdurre cambiamenti sostanziali nel software in fasi avanzate dello sviluppo abbia costi troppo elevati. Ciò che ne consegue è che ogni fase deve essere svolta (e dunque definita) in maniera esaustiva prima di passare alla successiva, in modo tale da non generare retroazioni.
Ogni fase produce in uscita dei **semilavorati**, che verranno presi in ingresso dalla fase successiva. Per questo motivo è fondamentale definire in maniera precisa sia come devono essere fatti i semilavorati che le date entro le quali i semilavorato devono essere prodotti.
Il limite più grosso del modello a cascata è la sua **rigidità**. In particolare, risultano rigidi i due principi su cui si basa il modello:

- **Immutabilità dell'analisi:** questo principio si basa sull'assunto che i clienti siano in grado sin da subito di esprimere in maniera chiara le loro esigenze e, di conseguenza, in fase di analisi iniziale è possibile definire tutte le funzionalità che il software deve realizzare. Nella realtà poi le specifiche fornite dal cliente possono cambiare via via che il sistema prende forma.
- **Immutabilità del progetto**, secondo il quale è possibile progettare l'intero sistema prima di aver scritto una sola riga di codice. In realtà spesso alcune idee vengono in mente solo quando si va a scrivere il codice, e comunque a volte è necessario rimettere mano alla progettazione per problemi legati alle prestazioni.

Questi problemi possono essere risolti provando a elaborare un prototipo prima di iniziare a lavorare sul sistema vero e proprio, anche se poi questo comporta una spesa tale da annullare i benefici portati dal modello a cascata.

Pattern MVC

Il **pattern MVC** è un pattern architetturale in grado di separare la logica di presentazione dalla logica di business. L'applicazione viene suddivisa in tre parti:

- **Modello:** rappresenta il livello dei dati, gestendo le operazioni di accesso e modifica. Quando il modello viene modificato viene generato un evento e le view associate vengono notificate. Generalmente è un **Observable**
- **View:** si occupa di mostrare all'utente una rappresentazione visuale dei dati di modello su un dispositivo di output. Si registra presso il modello per ricevere l'evento di cambiamento di stato e aggiorna la presentazione dei dati di conseguenza. Generalmente è un **Observer**
- **Controller:** definisce il comportamento dell'applicazione, gestendo gli input dell'utente e mappando le azioni dell'utente in comandi. I comandi sono poi inviati al modello o alla view

Dependency Inversion Principle

Il principio di **inversione delle dipendenze** (o *dependency inversion principle*) afferma che i moduli di alto livello, ovvero i *clienti*, non dovrebbero dipendere direttamente dai moduli di basso livello, ovvero i *fornitori di servizi*, ma entrambi dovrebbero dipendere da **astrazioni**. Tendenzialmente i moduli di basso livello, siccome sono i moduli con la maggior mole di codice, sono anche quelli più soggetti ai cambiamenti. Se i requisiti cambiano o è necessario correggere un errore e si modificano i moduli di basso livello, tutti i moduli che ne dipendono devono essere modificati, causando:

- **rigidità** → bisogna intervenire su un alto numero di moduli
- **fragilità** → si ha la potenzialità di introdurre errori in più parti del sistema
- **immobilità** → non si può riutilizzare il codice perché porta con sé troppe dipendenze.

Utilizzando il principio dell'inversione delle dipendenze, i moduli di alto livello dipenderebbero solamente dalle interfacce definite precedentemente in analisi, rendendo possibile una ri-fattorizzazione dei moduli di basso livello.
Questo principio riesce a funzionare perchè:

- le **astrazioni** contengono pochissimo codice (in teoria nulla, sono interfacce o classi astratte) e quindi sono raramente soggette a cambiamenti;
- i moduli **non astratti** sono soggetti a cambiamenti ma, siccome che nessuno dipende direttamente da questi moduli, le modifiche sono sicure

[EXTRA] I dettagli del sistema sono stati isolati, separati da un muro di astrazioni stabili, e nessuna dipendenza vincola le modifiche ad altri componenti: questo

- impedisce ai cambiamenti di propagarsi (**design for change**);
- consente un maggior riuso dei componenti perché questi sono fortemente disaccoppiati fra loro (**design for reuse**).

Pattern MVP

Il pattern MVP riprende la filosofia di fondo del pattern MVC, ma separando ulteriormente i ruoli dei singoli componenti. In particolare con questo pattern si fa uso di più Presenter, ciascuno assegnato ad una View.

- **Modello**: gestisce un insieme di dati logicamente correlati; risponde alle interrogazioni sui dati; risponde alle istruzioni di modifica dello stato; genera un evento quando lo stato cambia; registra, in forma anonima, gli oggetti interessati alla notifica dell'evento.
- **View**: gestisce un'area di visualizzazione, nella quale presenta all'utente una vista dei dati ricevuti dal presenter; mappa i dati forniti dal presenter, o una parte, in oggetti visuali e mostra tali oggetti su un particolare dispositivo di output; si registra presso il presenter per ricevere l'evento di cambiamento di stato.
- **Presenter**: gestisce gli input dell'utente (mouse, tastiera, etc.); mappa le azioni dell'utente nei comandi; invia tali comandi al model, che effettua le operazioni appropriate; aggiorna la view;

Interface Segregation Principle

Il **principio di segregazione delle interfacce** dice che:

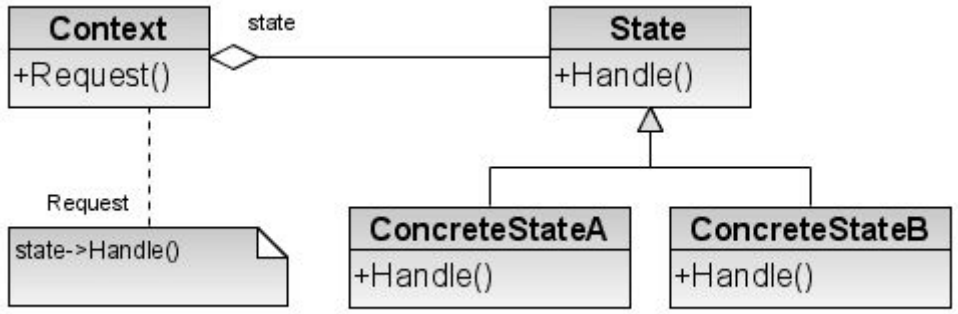
- I clienti non dovrebbero essere forzati a dipendere da interfacce che non usano
- Più interfacce specifiche sono meglio di una unica generale

Le interfacce troppo generali, dette **fat interfaces**, creano una forma indiretta di accoppiamento fra i clienti: se un cliente richiede l'aggiunta di una nuova funzionalità all'interfaccia, ogni altro cliente è costretto a cambiare anche se non è interessato alla nuova funzionalità. Questo crea un inutile sforzo di manutenzione e può rendere difficile trovare eventuali errori.

Se i servizi di una classe possono essere suddivisi in gruppi e ogni gruppo viene utilizzato da un diverso insieme di clienti, occorre creare interfacce specifiche per ogni tipo di cliente e implementare tutte le interfacce nella classe.

Pattern state **(behavioral - comportamentale)**

Problema: Il pattern State viene utilizzato quando si vuole che un oggetto cambi comportamento al cambiare del suo stato.
Soluzione: Per risolvere il problema si fa uso di un meccanismo di delega, grazie al quale l'oggetto è in grado di comportarsi come se avesse cambiato classe. Si associa il contesto a una classe astratta AbstractState, che rappresenta lo stato. AbstractState è implementata dai vari ConcreteState, che hanno un metodo Handle(). A ogni cambiamento di stato, il contesto chiama Request() che, per implementare il comportamento che dipende dallo stato, a sua volta chiama Handle() del proprio stato corrente.
In questo modo viene localizzato il comportamento specifico di uno stato e il comportamento è suddiviso in funzione dello stato.
Il pattern state permette anche di emulare l'ereditarietà multipla.
UML:



Interfaccia vs classe astratta

Interfaccia	Classe astratta
Deve descrivere una funzionalità semplice, implementabile da oggetti eterogenei (cioè appartenenti a classi non correlate tra loro) Ad esempio ● le istanze di tutte le classi che implemnetano l'interfaccia ICloneable sono clonabili ● le istanze di tutte le classi che implemnetano l'interfaccia IList sono trattate come collezioni	Può descrivere una funzionalità anche complessa, comune a un insieme di oggetti omogenei (cioè appartenenti a classi strettamente correlate tra di loro) Ad esempio: ● la classe astratta Enum fornisce funzionalità di base di tutti i tipi enumerativi
Può estendere 0+ interfacce	Può ereditare ● da 0+ interfacce ● da 0+ classi ○ minimo 1 se esiste una classe radice del sistema (eg Object) ○ massimo 1 se non è consentita ereditarietà multipla
Non può contenere uno stato	Può contenere uno stato
Non può contenere attributi membro e metodi statici	Può contenere attributi membro e metodi statici
Non contiene alcuna implementazione	Può essere implementata parzialmente, completamente o per nulla
Le classi concrete che la implementano devono realizzare tutte le funzionalità	Le classi concrete che la estendono: ● devono realizzare tutte le funzionalità non implementate ● possono fornire una realizzazione alternativa a quelle implementate
Deve essere stabile	Può essere modificata
Se si aggiungesse un metodo a un'interfaccia già in uso, tutte le classi che implementano quell'interfaccia dovrebbero essere modificate	Quando si aggiunge un metodo ad una classe già in uso, è possibile fornire un'implementazione di default in modo tale da non dover modificare le sottoclassi

Tecnologia COM

COM è un **sistema orientato agli oggetti**, distribuito e indipendente dalla piattaforma, ideato per creare componenti software binari.
Non è un linguaggio, bensì uno standard → specifica i requisiti di programmazione che permettono agli oggetti COM di interagire con gli altri oggetti

La tecnologia COM specifica l'uso della tecnica di **reference counting** al fine di assicurare che gli oggetti rimangano "vivi" solo finché ci sono dei clienti in grado di utilizzarli. Un oggetto COM è responsabile della deallocazione della propria memoria, una volta che il suo contatore dei riferimenti arriva a 0.

La posizione di ogni componente è salvata nel registro Windows, quindi ci può essere solo una specifica versione di un determinato componente → limitazioni nel deployment, non possono esistere componenti nel sistema che usano versioni diverse dello stesso componente

Framework .NET

Ambiente di esecuzione + libreria di classi → semplifica lo sviluppo e il deployment.
Fortemente integrato con COM ma ne è completamente indipendente.
.NET è un'implementazione di CLI (Common Language Infrastructure)
Concetti chiave di .NET:

- **IL** → Intermediate Language
- **CLR** → Common Language Runtime, ambiente di esecuzione runtime per le applicazioni .NET
- **CTS** → Common Type System, tipi di dato supportati da .NET
    - Progettato per linguaggi object oriented, procedurali e funzionali
    - Alla base: *classi*, *strutture*, *interfacce*, *enumerativi*, *delegati*
    - Fortemente tipizzato (a tempo di compilazione)
    - *Late binding*
    - Ereditarietà
        - singola di estensione
        - multipla di interfaccia
- **CLS** → Common Language Specification, è un sottoinsieme di CTS che definisce le regole che i linguaggi devono seguire per essere interoperabili all'interno di .NET
    - Regole per gli identificatori (case sensitivity, keywords…)
    - Regole per costruzione oggetti, per denominazione proprietà...

Processo di compilazione:
Sorgenti → Compilatori .NET → Codice IL (Assembly) → Compilatore JIT → Codice nativo → esecuzione

Pattern flyweight (structural - strutturale)

Il pattern flyweight è un Design pattern che descrive come condividere oggetti leggeri, ovvero a granularità molto fine, in modo tale che il loro uso non sia troppo costoso.
Si definisce *flyweight* un **oggetto condiviso** il cui uso, da parte di più clienti, può essere simultaneo ed efficiente. Nonostante sia condiviso, **non deve essere distinguibile da oggetti non condivisi**.
Gli oggetti flyweight non devono poter essere istanziati direttamente dal cliente, ma devono essere ottenuti esclusivamente da una **FlyweightFactory**.

Esempio codice C#:

```
private Dictionary<KeyType, FlyweightType> flyweights;
…
public FlyweightType GetFlyweight(KeyType key)
{
      if(!flyweights.ContainsKey(key))
      {
            flyweights.Add(key, CreateFlyweight(key));
      }
      return flyweights[key];
}
```

Per ogni oggetto si può attuare la distinzione tra:

- **stato intrinseco:** non dipende dal contesto di utilizzo e quindi **può essere condiviso** (memorizzato nel flyweight)
- **stato estrinseco:** è dipendente dal contesto di utilizzo, quindi **non può essere condiviso** (memorizzato nel cliente, viene passato al flyweight quando viene invocata una sua operazione)